# Consistency and Replication

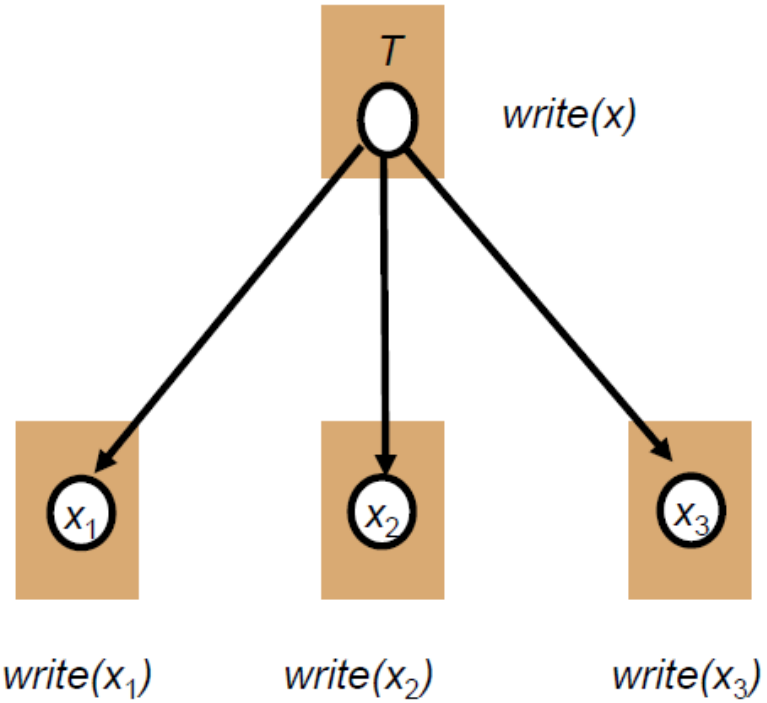
Distributed Systems [7]

# Replication

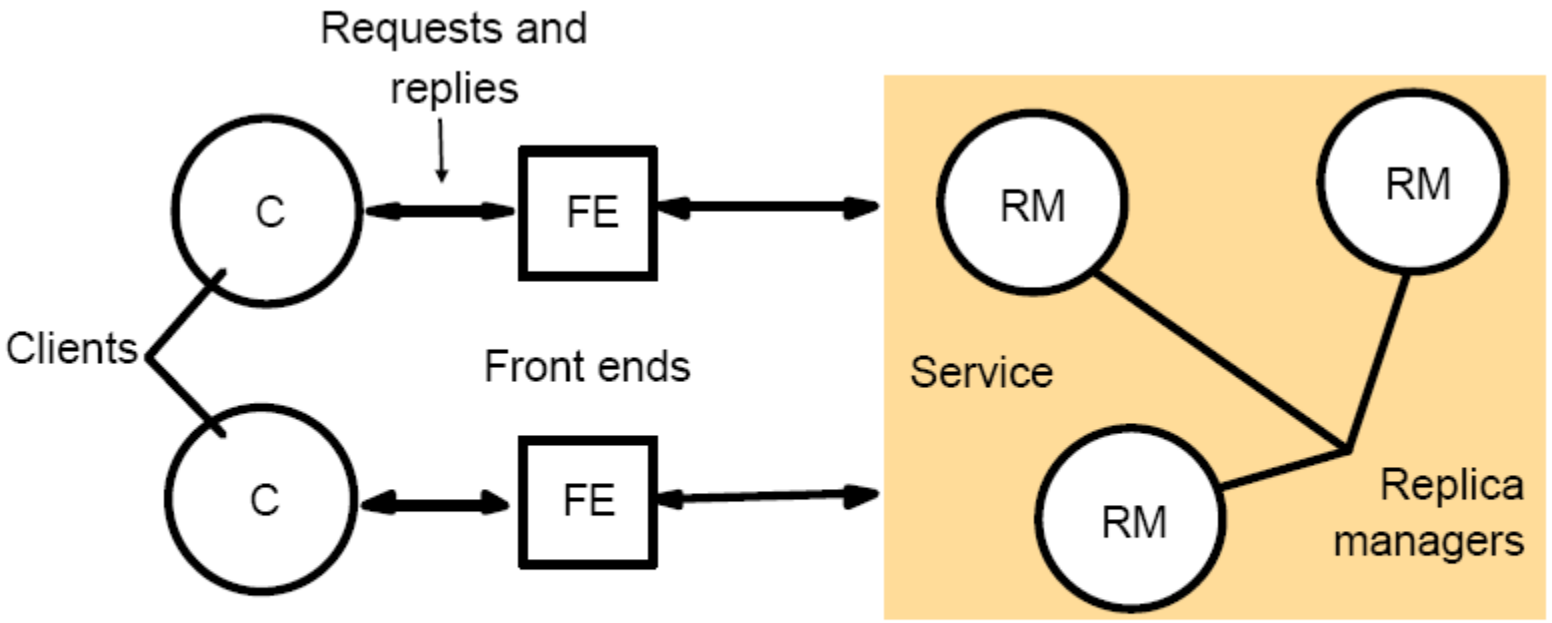
- Why replicate?
  - Reliability
    - Avoid single points of failure
  - Performance
    - Scalability in numbers and geographic area
- Why not replicate?
  - Replication transparency
  - Consistency issues
    - Updates are costly
    - Availability *may* suffer if not careful

# Logical vs Physical Objects

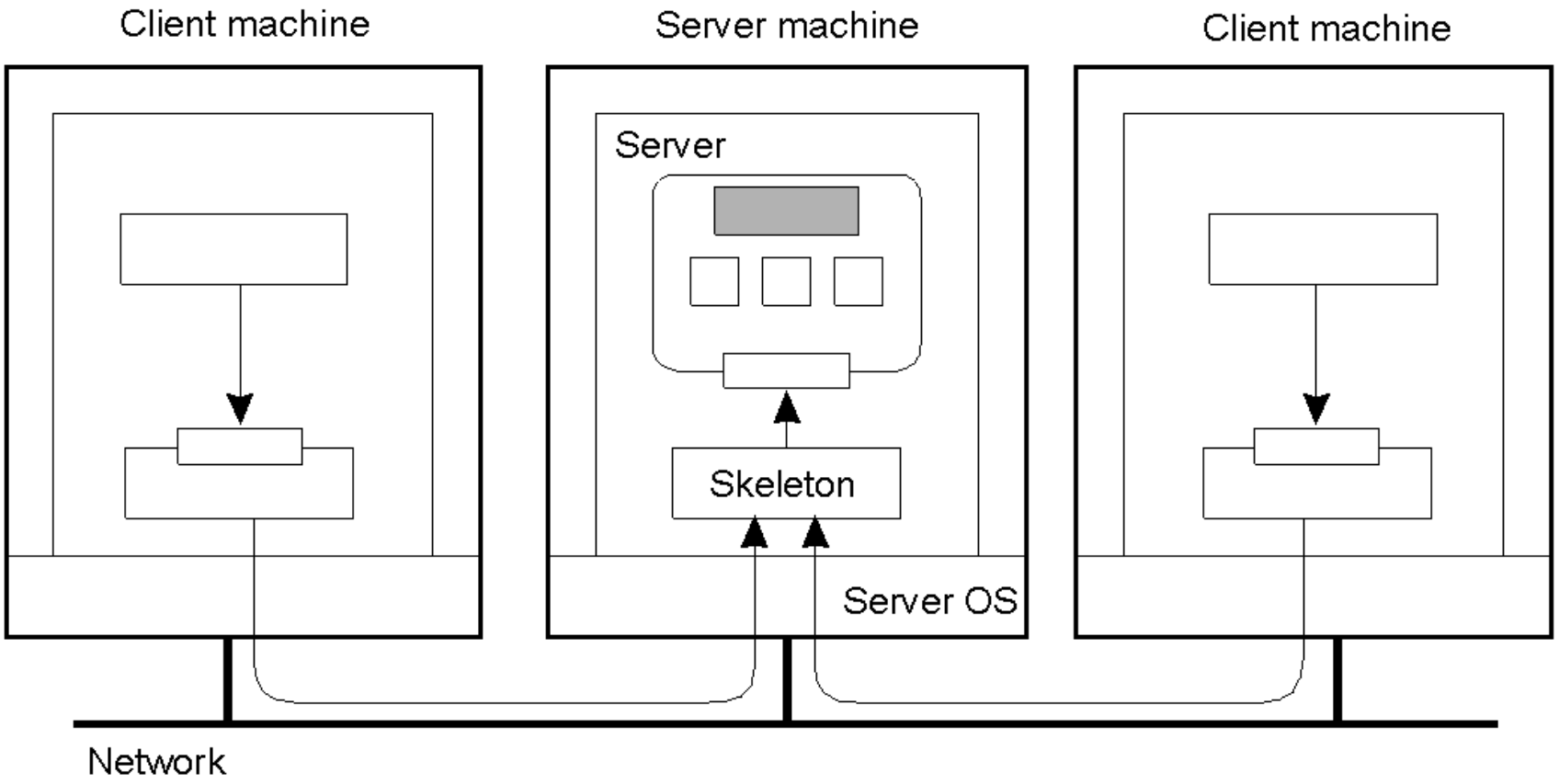
- There are physical copies of logical objects in the system.
  - Operations are specified on logical objects, but translated to operate on physical objects.



# Replication Architecture

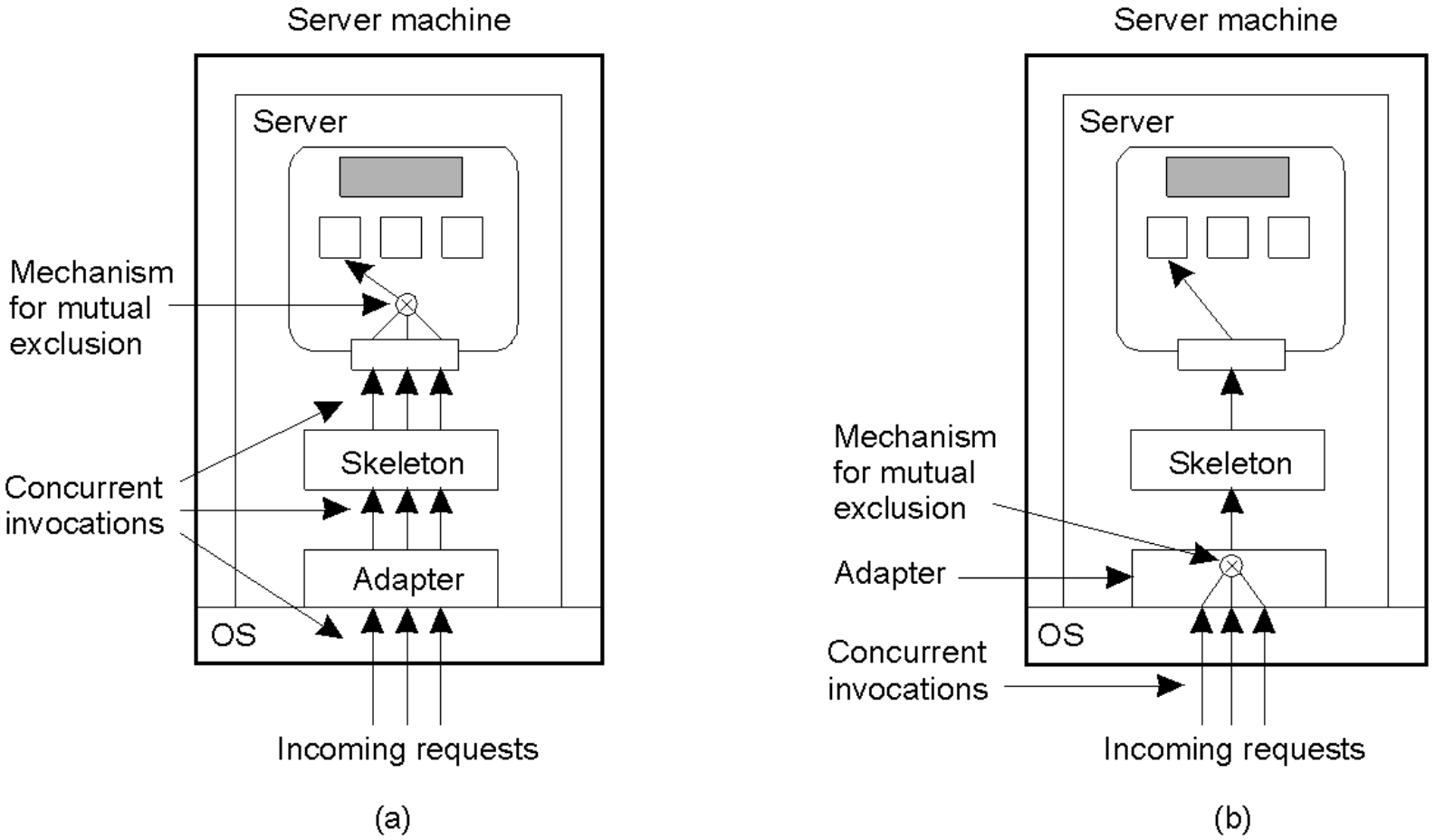


# Object Replication (1)




- Organization of a distributed remote object shared by two different clients.

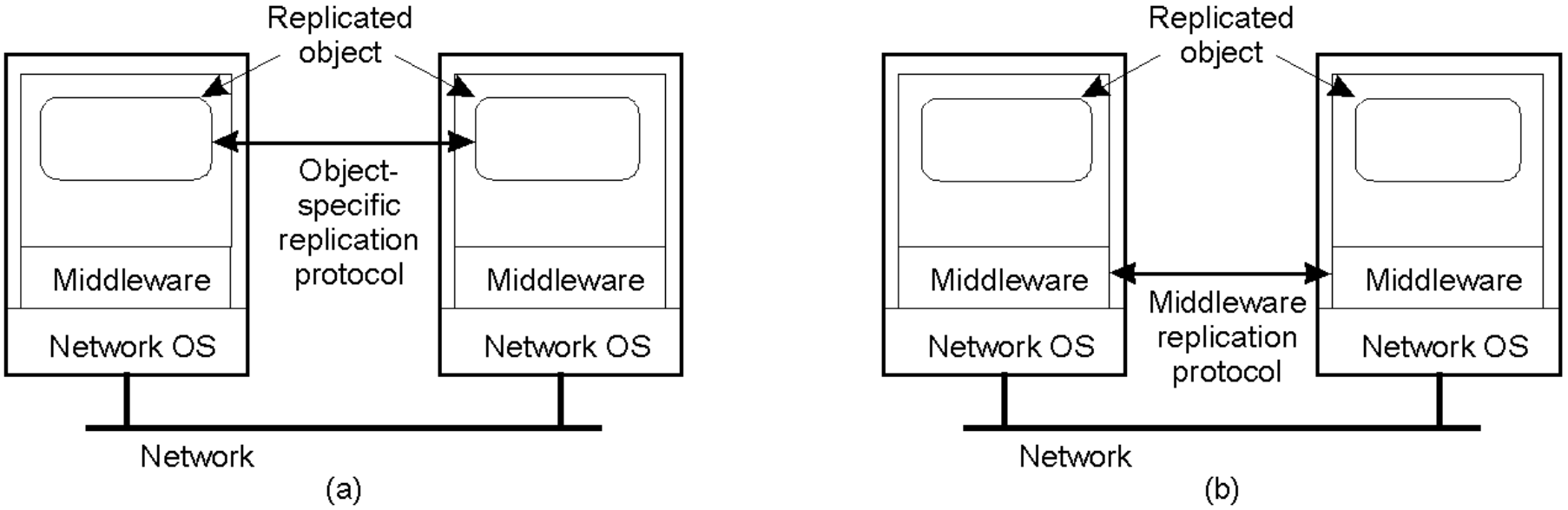
# Object Replication (2)




a) A remote object capable of handling concurrent invocations on its own.

b) A remote object for which an object adapter is required to handle concurrent invocations

# Object Replication (3)




a) A distributed system for replication-aware distributed objects.

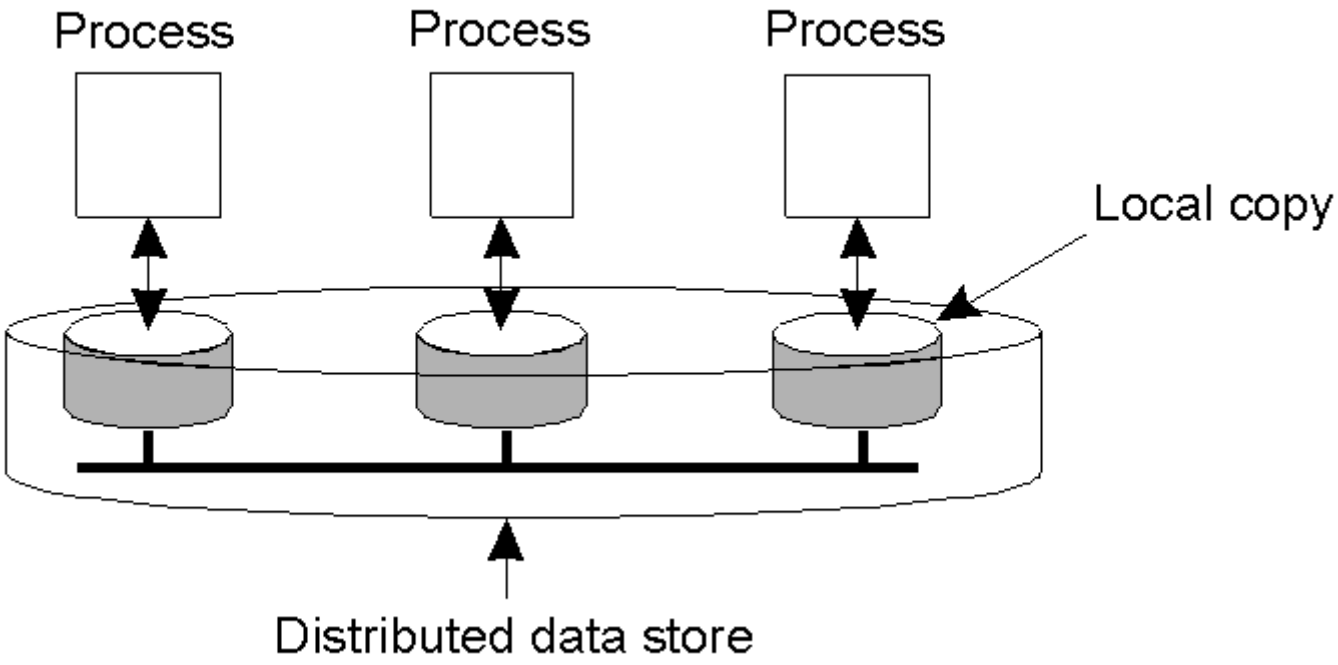
b) A distributed system responsible for replica management

# What will we study

- Consistency models - How do we reason about the consistency of the "global state"?
  - Data-centric consistency
    - Strict consistency
    - Sequential consistency
    - Linearizability
  - Client-centric consistency
    - Eventual consistency
- Update propagation - How does an update to one copy of an item get propagated to other copies?
- Replication protocols - What is the algorithm that takes one update propagation method and enforces a given consistency model?

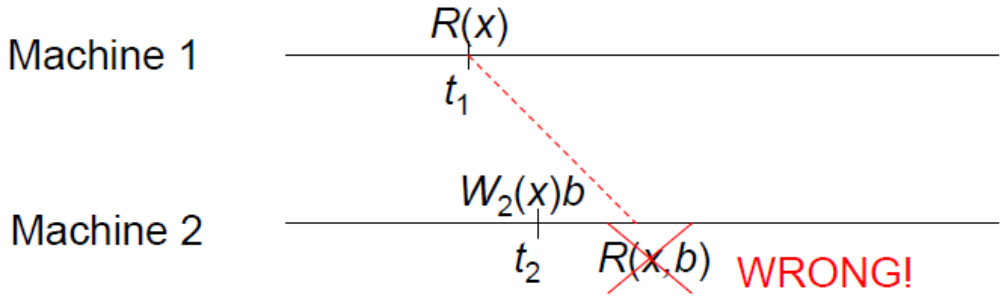
# Data-Centric Consistency Models

- The general organization of a logical data store, physically distributed and replicated across multiple processes.

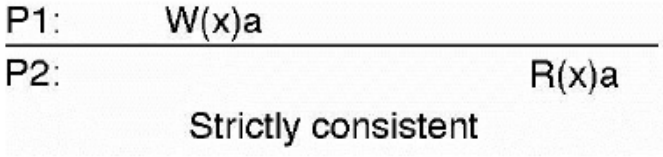


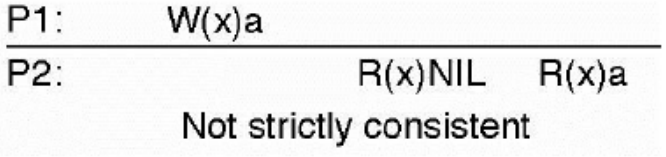
# Strict Consistency

- Any *read*(*x*) returns a value corresponding to the result of the most recent *write*(*x*).



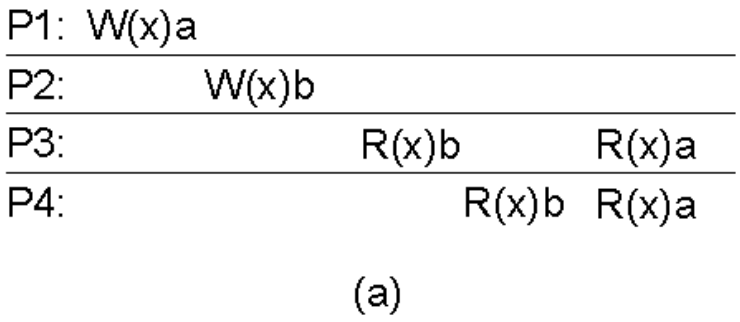

- Relies on absolute global time; all writes are instantaneously visible to all processes and an absolute global time order is maintained.
- Cannot be implemented in a distributed system



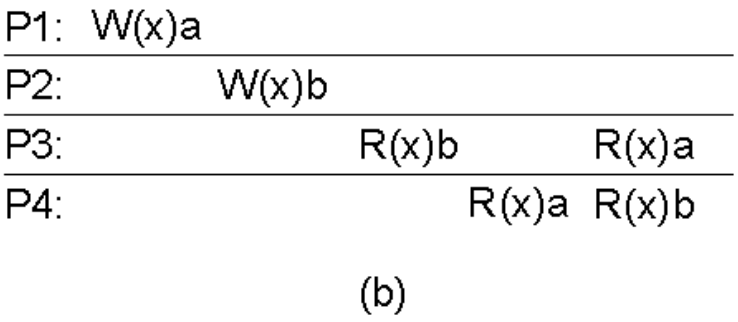



# Sequential Consistency

- Similar to linearizability, but no requirement on timestamp order.
- The result of execution should satisfy the following criteria:
  - Read and write operations by all processes on the data store were executed in some sequential order;
  - Operations of each individual process appear in this sequence in the order specified by its program.
- These mean that all processes see the same interleaving of operations similar to serializability.




(a)




(b)

# Linearizability

- The result of the execution should satisfy the following criteria:
  - Read and write by all processes were executed in some serial order and each process's operations maintain the order of specified;
  - If $ts_{op1}(x) < ts_{op2}(y)$ then $op1(x)$ should precede $op2(y)$ in this sequence. This specifies that the order of operations in itnerleaving is conssitent with the real times at which the operations occurred in the actual implementation.
- Requires synchronization according to timestamps, which makes it expensive.
- Used only in formal verification of programs.

# Sequences for the Processes

| Process P1 | Process P2 | Process P3 |
|---|---|---|
| x = 1; | y = 1; | z = 1; |
| print ( y, z); | print (x, z); | print (x, y); |

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

| | | | |
|---|---|---|---|
| x = 1; | x = 1; | y = 1; | y = 1; |
| print ((y, z); | y = 1; | z = 1; | x = 1; |
| y = 1; | print (x,z); | print (x, y); | z = 1; |
| print (x, z); | print(y, z); | print (x, z); | print (x, z); |
| z = 1; | z = 1; | x = 1; | print (y, z); |
| print (x, y); | print (x, y); | print (y, z); | print (x, y); |
| Prints: 001011 | Prints: 101011 | Prints: 010111 | Prints: 111111 |
| (a) | (b) | (c) | (d) |

# Causal Consistency (1)

- Necessary condition:
  Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

# Causal Consistency (2)

| | | | | | |
|---|---|---|---|---|---|
| P1: | W(x)a | | W(x)c | | |
| P2: | | R(x)a | W(x)b | | |
| P3: | | R(x)a | | R(x)c | R(x)b |
| P4: | | R(x)a | | R(x)b | R(x)c |

- This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

# Causal Consistency (3)

| | | | | |
|---|---|---|---|---|
| P1: W(x)a | | | | |
| P2: | R(x)a | W(x)b | | |
| P3: | | | R(x)b | R(x)a |
| P4: | | | R(x)a | R(x)b |

(a)

| | | | | |
|---|---|---|---|---|
| P1: W(x)a | | | | |
| P2: | | W(x)b | | |
| P3: | | | R(x)b | R(x)a |
| P4: | | | R(x)a | R(x)b |

(b)

a) A violation of a causally-consistent store.

b) A correct sequence of events in a causally-consistent store.

# FIFO Consistency (1)

- Necessary Condition:
  Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

# FIFO Consistency (2)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| P1: W(x)a | | | | | | | |
| P2: | | R(x)a | | W(x)b | W(x)c | | |
| P3: | | | | | | R(x)b | R(x)a | R(x)c |
| P4: | | | | | | R(x)a | R(x)b | R(x)c |

- A valid sequence of events of FIFO consistency

# FIFO Consistency (3)

| Process P1 | Process P2 | Process P3 |
|---|---|---|
| x = 1; | y = 1; | z = 1; |
| print ( y, z); | print (x, z); | print (x, y); |

- The statements in bold are the ones that generate the output shown.

| | | |
|---|---|---|
| x = 1; | x = 1; | y = 1; |
| **print (y, z);** | y = 1; | print (x, z); |
| y = 1; | **print(x, z);** | z = 1; |
| print(x, z); | print ( y, z); | **print (x, y);** |
| z = 1; | z = 1; | x = 1; |
| print (x, y); | print (x, y); | print (y, z); |
| | | |
| Prints: 00 | Prints: 10 | Prints: 01 |
| (a) | (b) | (c) |

# FIFO Consistency (4)

| Process P1 | Process P2 |
| --- | --- |
| x = 1; | y = 1; |
| if (y == 0) kill (P2); | if (x == 0) kill (P1); |

- Two concurrent processes.

# Weak Consistency (1)

- Properties:
  - Accesses to synchronization variables associated with a data store are sequentially consistent
  - No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
  - No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

# Weak Consistency (2)

```
int a, b, c, d, e, x, y;          /* variables */
int *p, *q;                       /* pointers */
int f( int *p, int *q);           /* function prototype */

a = x * x;                        /* a stored in register */
b = y * y;                        /* b as well */
c = a*a*a + b*b + a * b;          /* used later */
d = a * a * c;                    /* used later */
p = &a;                           /* p gets address of a */
q = &b                            /* q gets address of b */
e = f(p, q)                       /* function call */
```

- A program fragment in which some variables may be kept in registers.

# Weak Consistency (3)

| | | | | | |
|---|---|---|---|---|---|
| P1: W(x)a | W(x)b | S | | | |
| P2: | | | R(x)a | R(x)b | S |
| P3: | | | R(x)b | R(x)a | S |

(a)

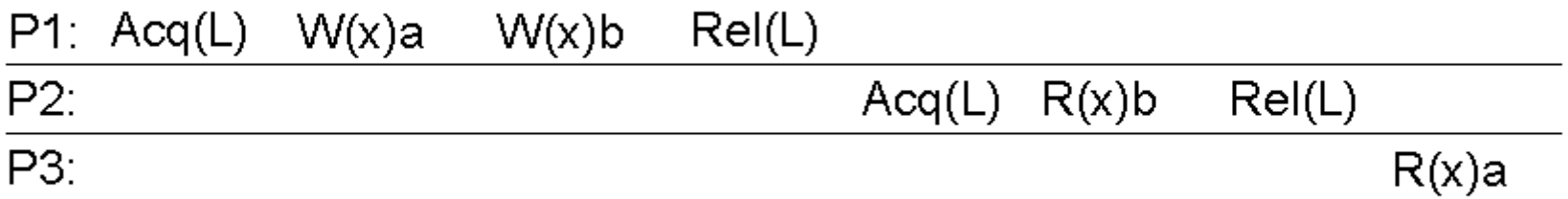
| | | | | |
|---|---|---|---|---|
| P1: W(x)a | W(x)b | S | | |
| P2: | | | S | R(x)a |

(b)

a) A valid sequence of events for weak consistency.

b) An invalid sequence for weak consistency.

# Release Consistency (1)




- A valid event sequence for release consistency.

# Release Consistency (2)

- Rules:
  - Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
  - Before a release is allowed to be performed, all previous reads and writes by the process must have completed
  - Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).

# Entry Consistency (1)

- Conditions:
  - An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
  - Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
  - After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

# Entry Consistency (2)

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P1: | Acq(Lx) | W(x)a | Acq(Ly) | W(y)b | Rel(Lx) | Rel(Ly) | | | | | | |
| P2: | | | | | | | Acq(Lx) | R(x)a | | | R(y)NIL | |
| P3: | | | | | | | | | Acq(Ly) | R(y)b | | |

- A valid event sequence for entry consistency.

# Summary of Consistency Models

| Consistency | Description |
|---|---|
| Strict | Absolute time ordering of all shared accesses matters. |
| Linearizability | All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp |
| Sequential | All processes see all shared accesses in the same order. Accesses are not ordered in time |
| Causal | All processes see causally-related shared accesses in the same order. |
| FIFO | All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order |

(a)

| Consistency | Description |
|---|---|
| Weak | Shared data can be counted on to be consistent only after a synchronization is done |
| Release | Shared data are made consistent when a critical region is exited |
| Entry | Shared data pertaining to a critical region are made consistent when a critical region is entered. |

(b)

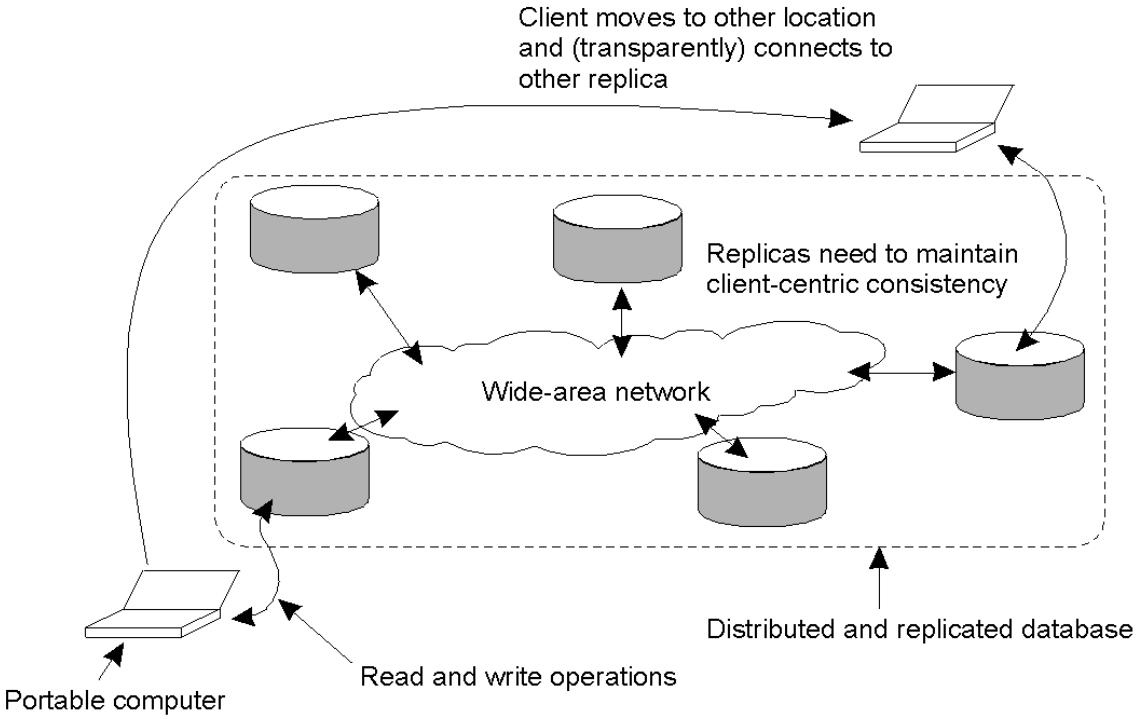
a) Consistency models not using synchronization operations.

b) Models with synchronization operations.

# Client-Centric Consistency

- More relaxed form of consistency → only concerned with replicas being eventually consistent (eventual consistency).
- In the absence of any further updates, all replicas converge to identical copies of each other → only requires guarantees that updates will be propagated.
- Easy if a user always accesses the same replica; problematic if the user accesses different replicas.
  - Client-centric consistency: guarantees for a single client the consistency of access to a data store.

# Eventual Consistency

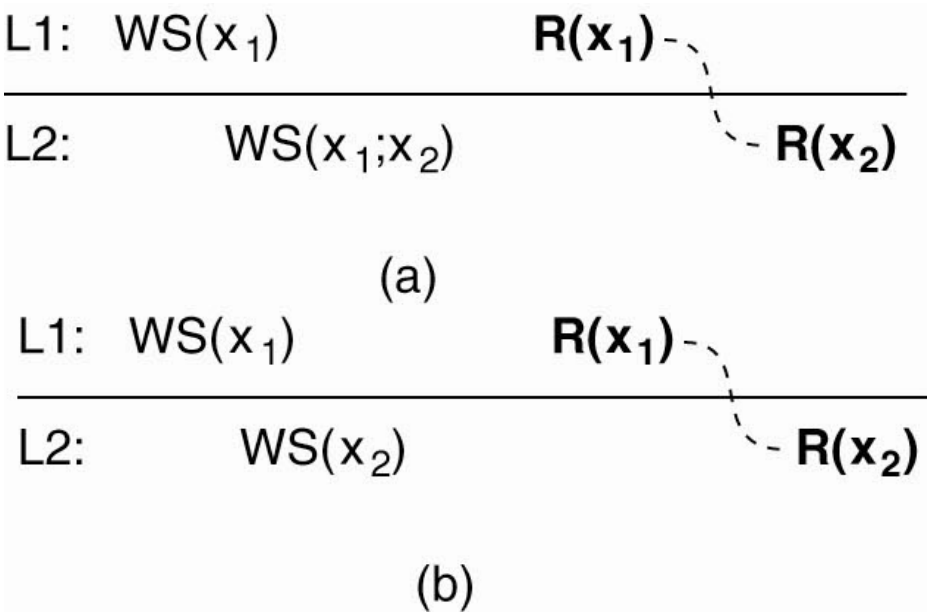



- The principle of a mobile user accessing different replicas of a distributed database.

# Client-Centric Consistency (2)

- Monotonic reads
  - If a process reads the value of a data item *x*, any successive read operation on *x* by that process will always return that same value or a more recent value.
- Monotonic writes
  - A write operation by a process on a data item *x* is completed before any successive write operation on *x* by the same process.
- Read your writes
  - The effect of a write operation by a process on data item *x* will always be seen by a successive read operation on *x* by the same process.
- Writes follow reads
  - A write operation by a process on a data item *x* following a previous read operation on *x* by the same process is guaranteed to take place on the same or more recent value of *x* that was read.

# Monotonic Reads

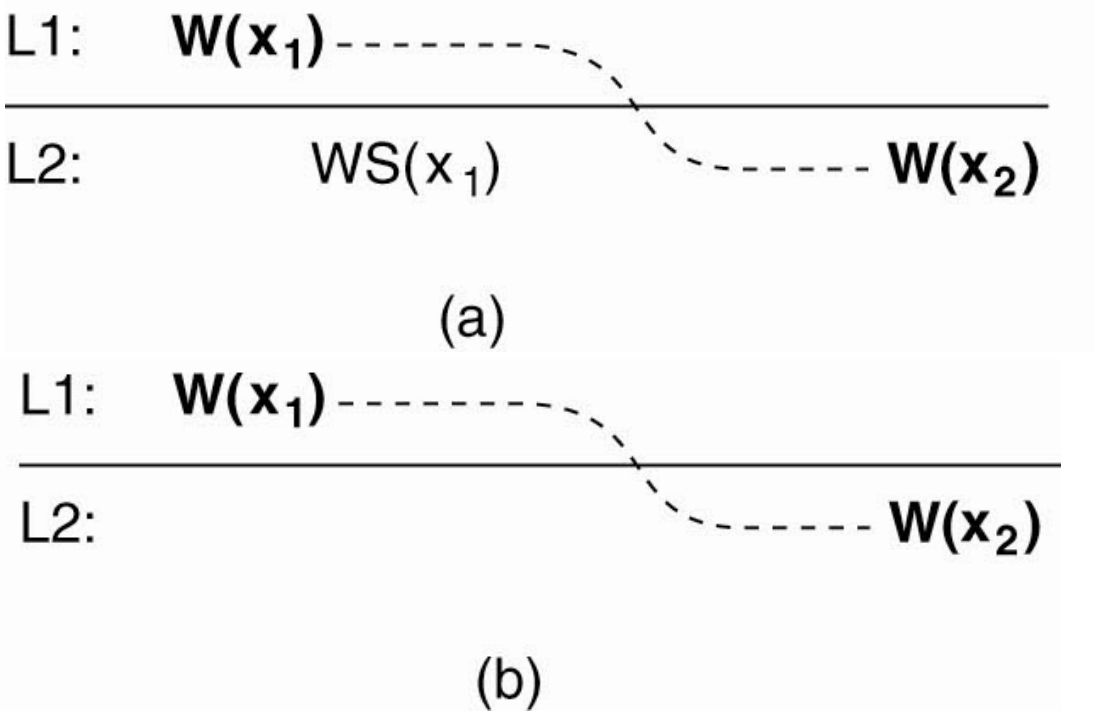



- The read operations performed by a single process *P* at two different local copies of the same data store.

a) A monotonic-read consistent data store

b) A data store that does not provide monotonic reads.

# Monotonic Writes

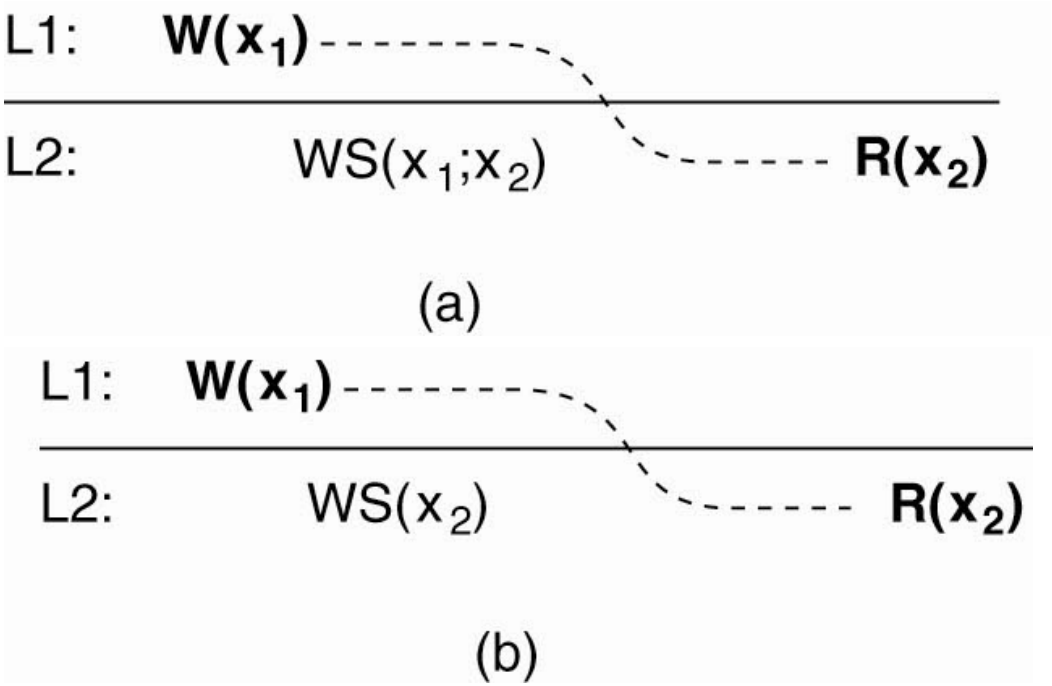



- The write operations performed by a single process *P* at two different local copies of the same data store

a) A monotonic-write consistent data store.

b) A data store that does not provide monotonic-write consistency.

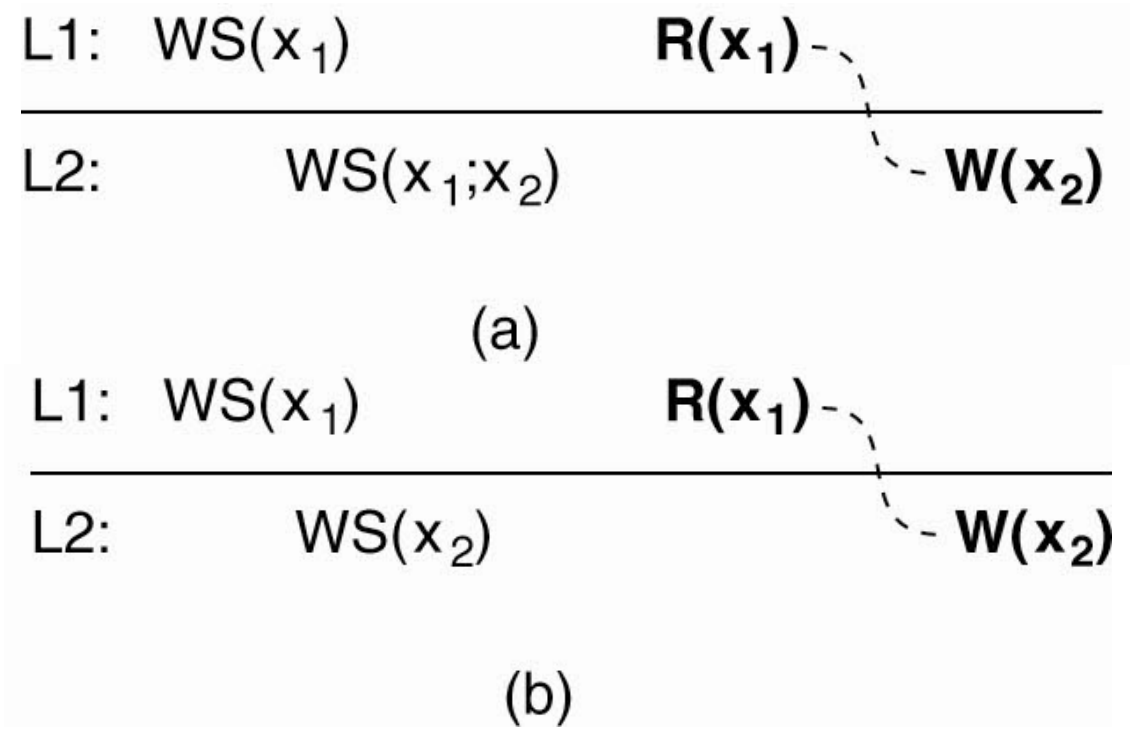
# Read Your Writes




a) A data store that provides read-your-writes consistency.

b) A data store that does not.

# Writes Follow Reads

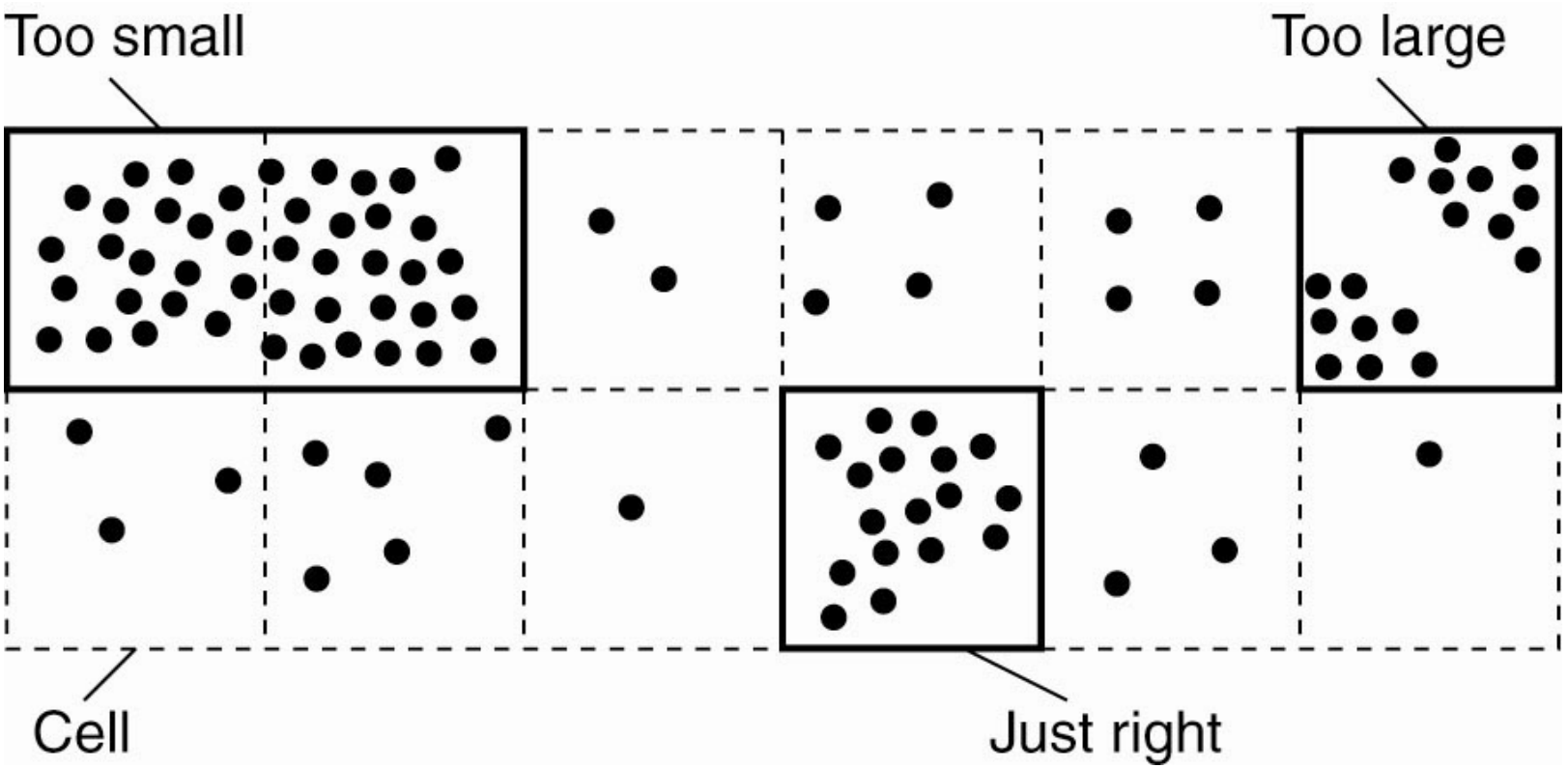



a) A writes-follow-reads consistent data store

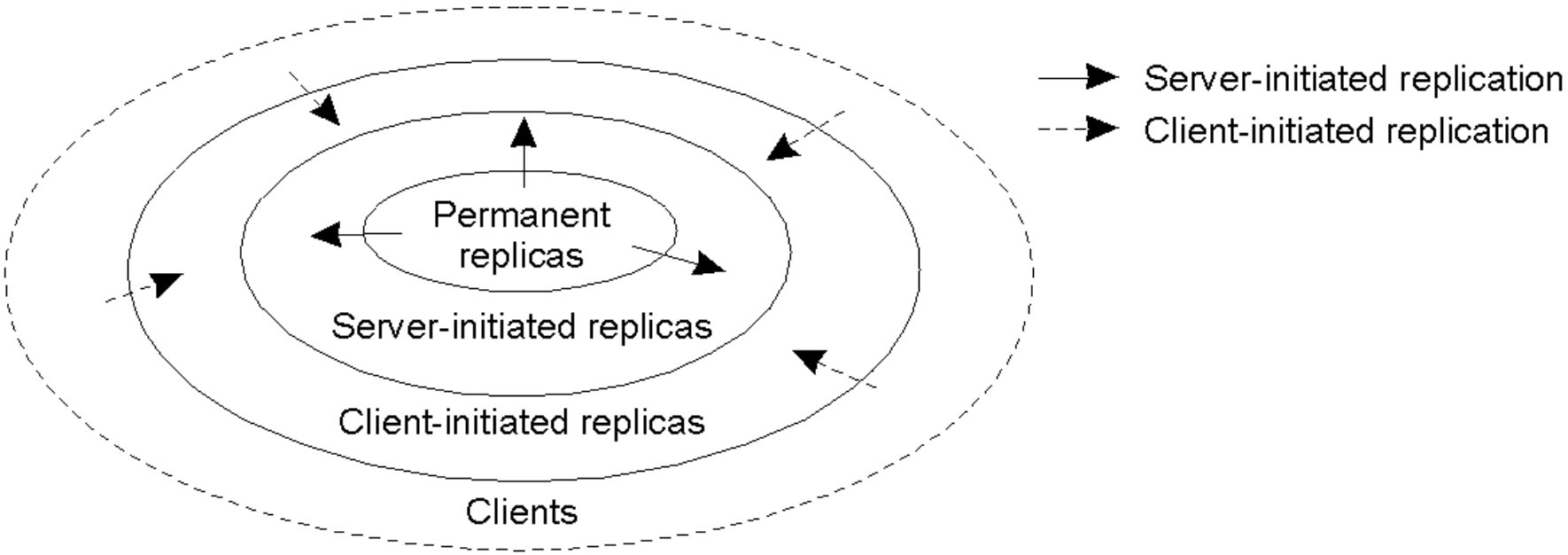
b) A data store that does not provide writes-follow-reads consistency

# Replica-Server Placement

- Choosing a proper cell size for server placement.



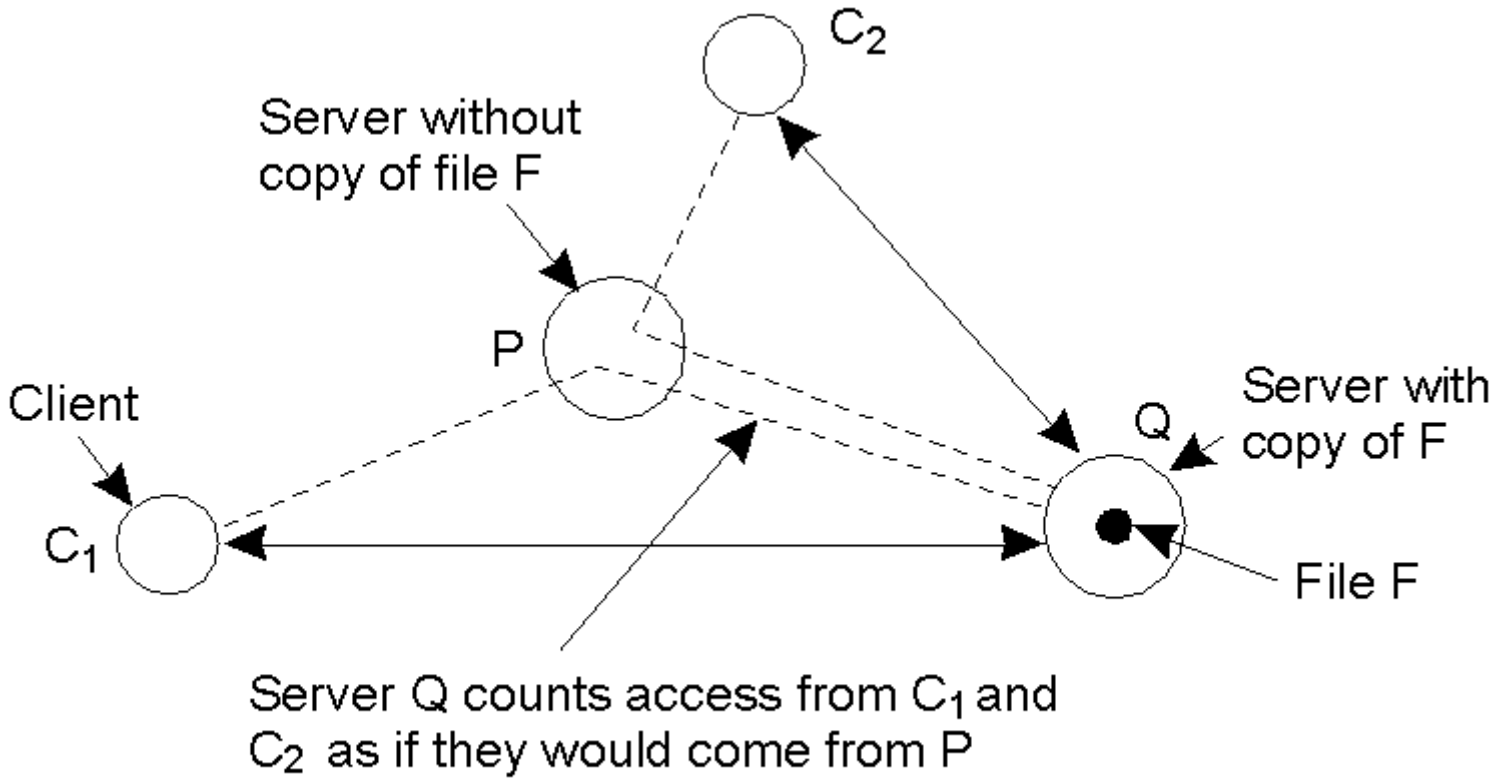
# Replica Placement




- The logical organization of different kinds of copies of a data store into three concentric rings.

# Server-Initiated Replicas

- Counting access requests from different clients.



# State versus Operations

- Possibilities for what is to be propagated:

1. Propagate only a notification of an update.
2. Transfer data from one copy to another.
3. Propagate the update operation to other copies.

# Pull vs. Push Protocols

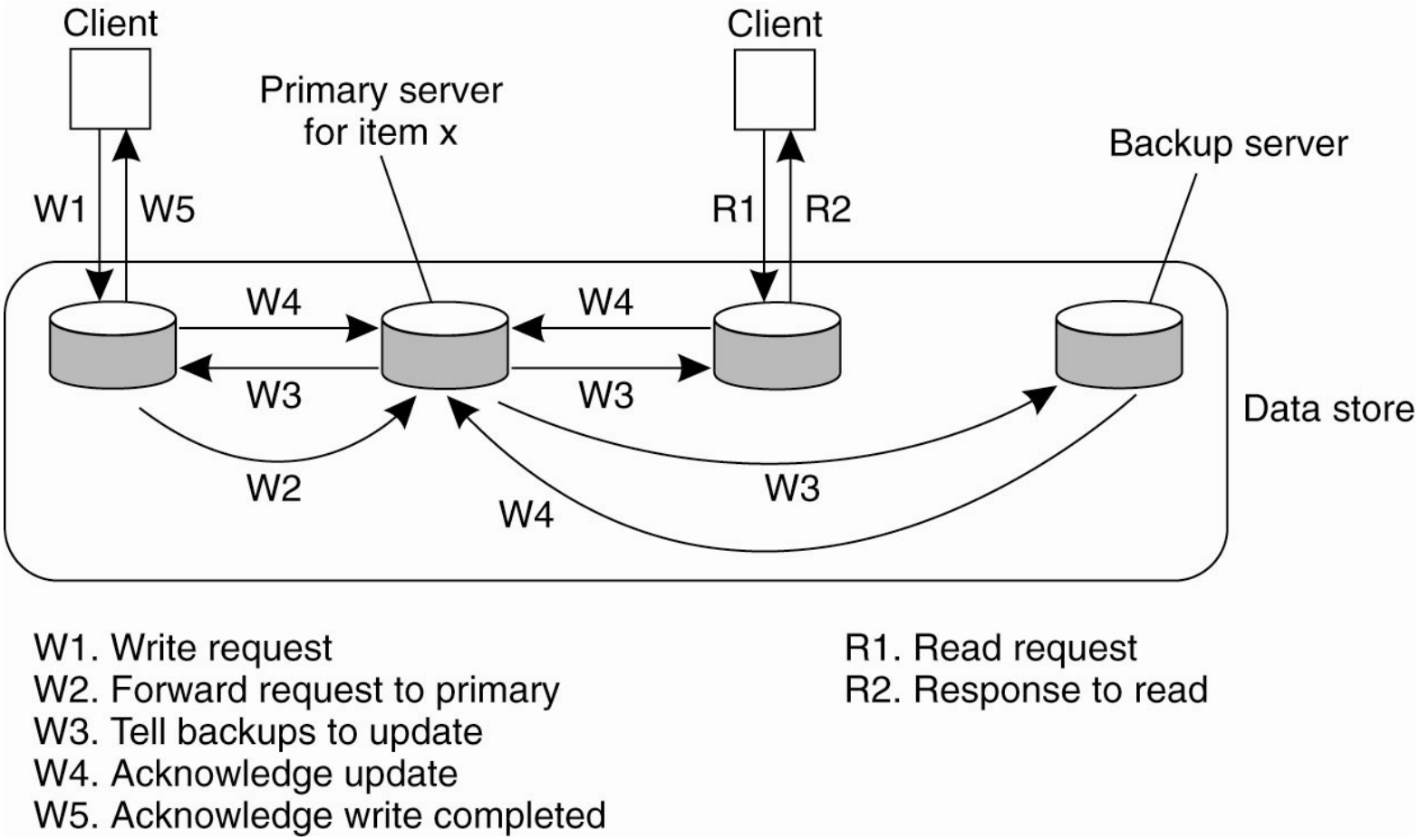
| Issue | Push-based | Pull-based |
|---|---|---|
| State of server | List of client replicas and caches | None |
| Messages sent | Update (and possibly fetch update later) | Poll and update |
| Response time at client | Immediate (or fetch-update time) | Fetch-update time |

- A comparison between push-based and pull-based protocols in the case of multiple client, single server systems.

# Replication Protocols

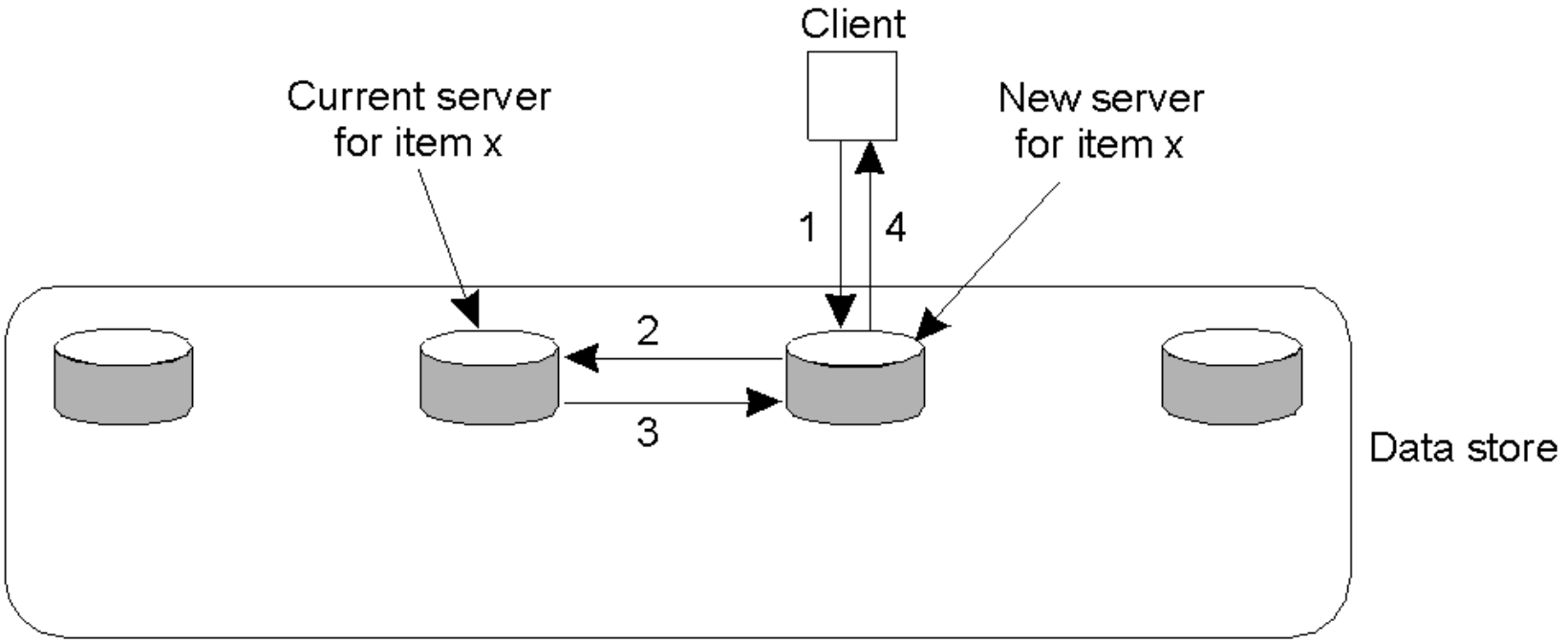
- Linearizability
  - Primary-Backup
  - Chain Replication
- Sequential consistency
  - Primary-based protocols
    - Remote-Write protocols
    - Local-Write protocols
  - Replicated Write protocols
    - Active replication
    - Quorum-based protocols

# Remote-Write Protocols




- Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded.

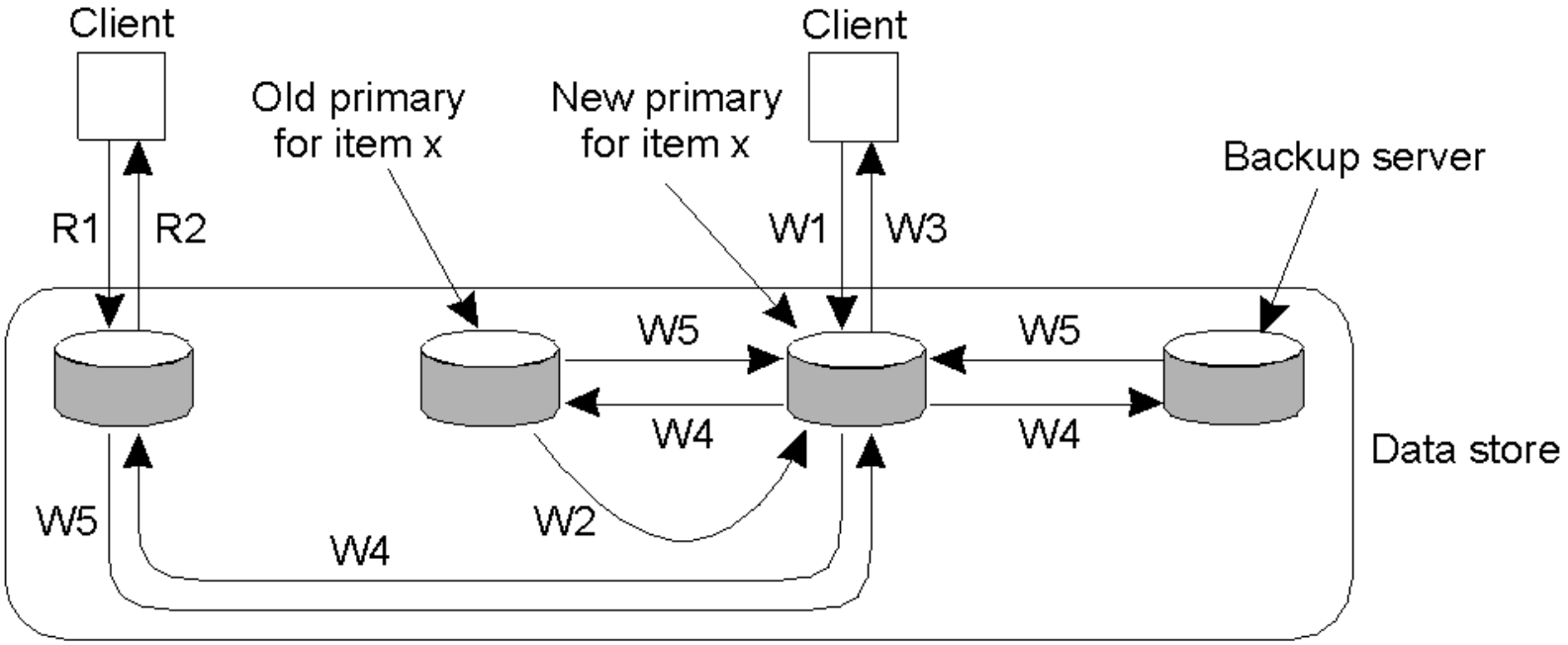
# Local-Write Protocols (1)




1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

- Primary-based local-write protocol in which a single copy is migrated between processes.

# Local-Write Protocols (2)




W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

- Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

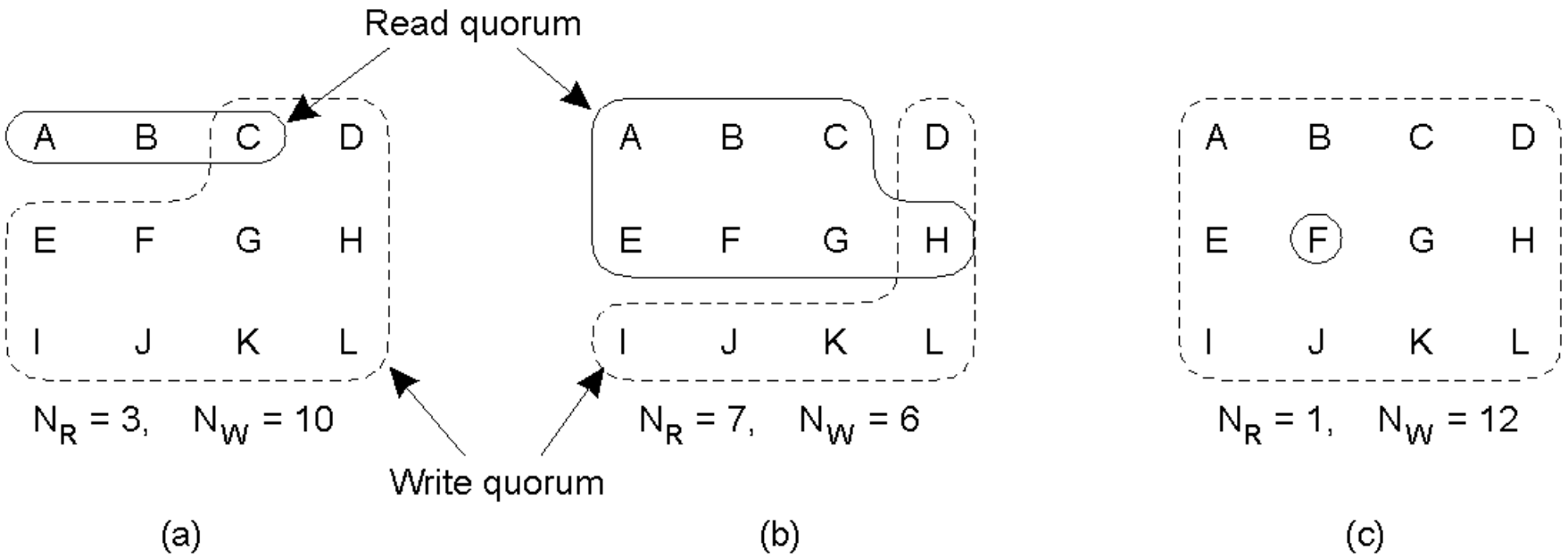
# Active Replication

- Requires a process, for each replica, that can perform the update on it
- How to enforce the update order?
  - Totally-ordered multicast mechanism needed
  - Can be implemented by Lamport timestamps
  - Can be implemented by sequencer
- Problem of replicated invocations
  - If an object *A* invokes another object *B*, all replicas of *A* will invoke *B* (multiple invocations)

# Quorum-Based Protocol

- Assign a vote to each *copy* of a replicated object (say *Vi*) such that $\sum_i V_i = V$
- Each operation has to obtain a *read quorum* ($V_r$) to read and a *write quorum* ($V_w$) to write an object
- Then the following rules have to be obeyed in determining the quorums:
  - $V_r + V_w > V$ — an object is not read and written by two transactions concurrently
  - $V_w > V/2$ — two write operations from two transactions cannot occur concurrently on the same object

# Quorum-Based Protocols



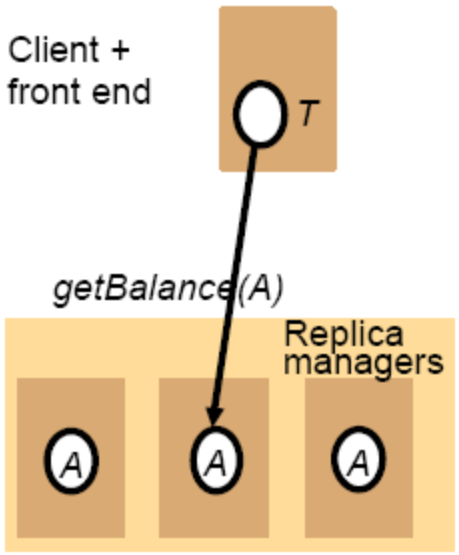

- Three examples of the voting algorithm:

a) A correct choice of read and write set

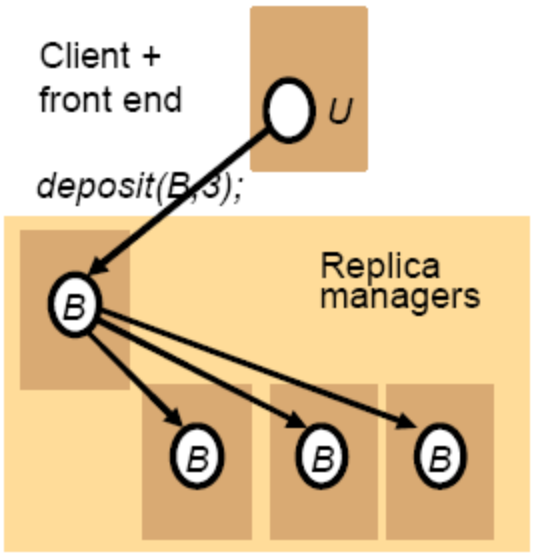
b) A choice that may lead to write-write conflicts

c) A correct choice, known as ROWA (read one, write all)

# ROWA

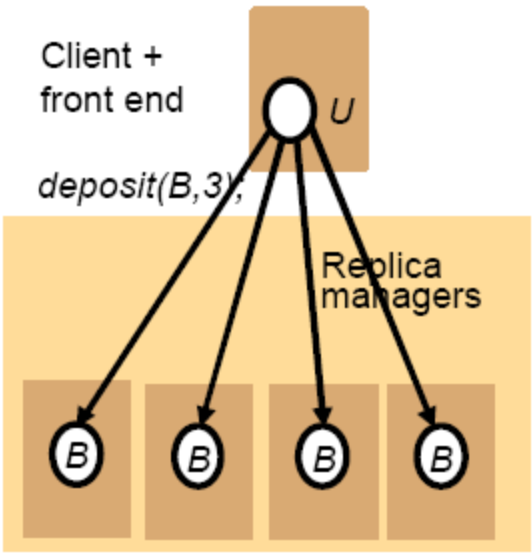



Read




Write with a primary copy

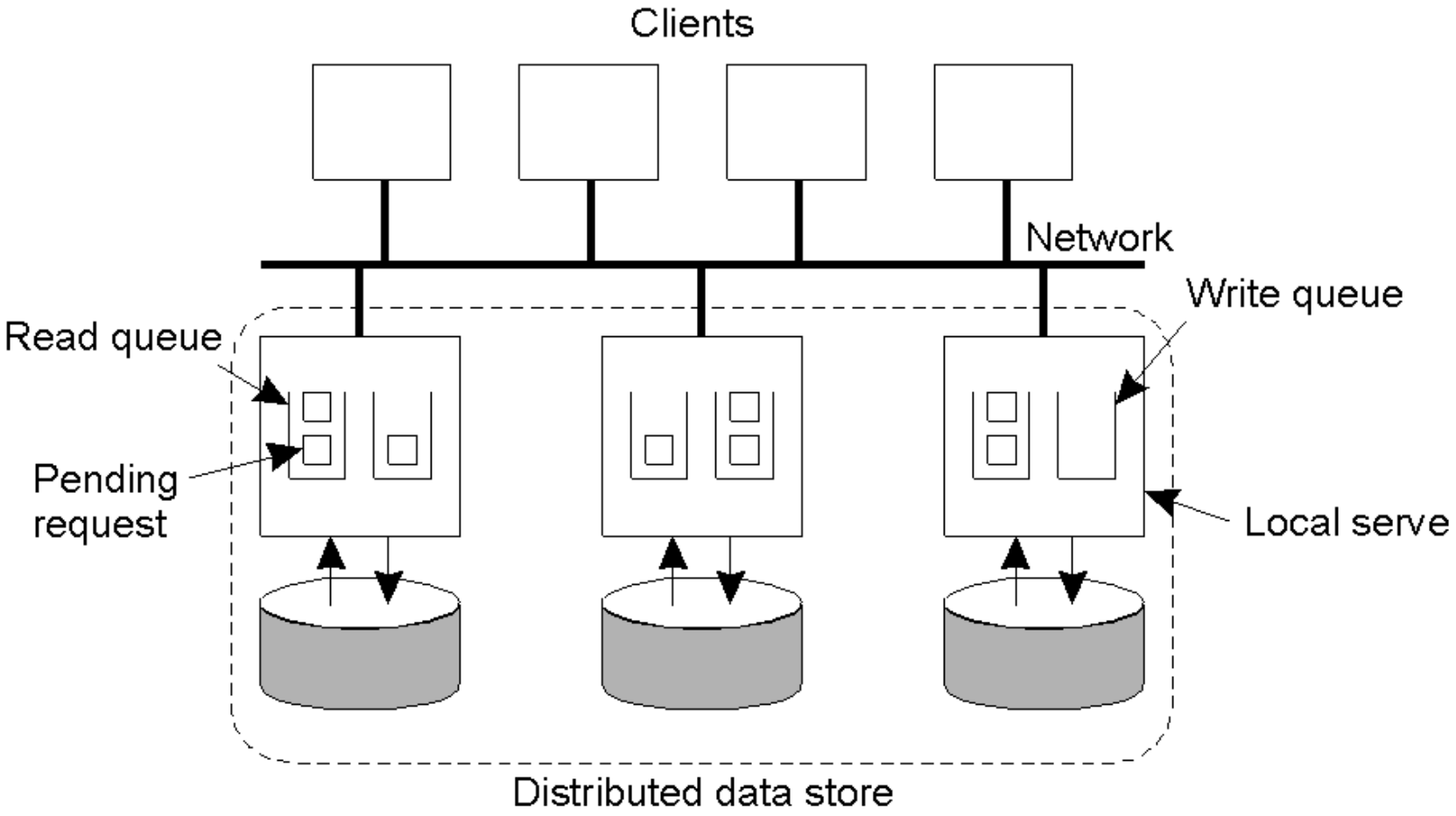
A 2 level nested 2PC protocol is implemented




Write without a primary copy

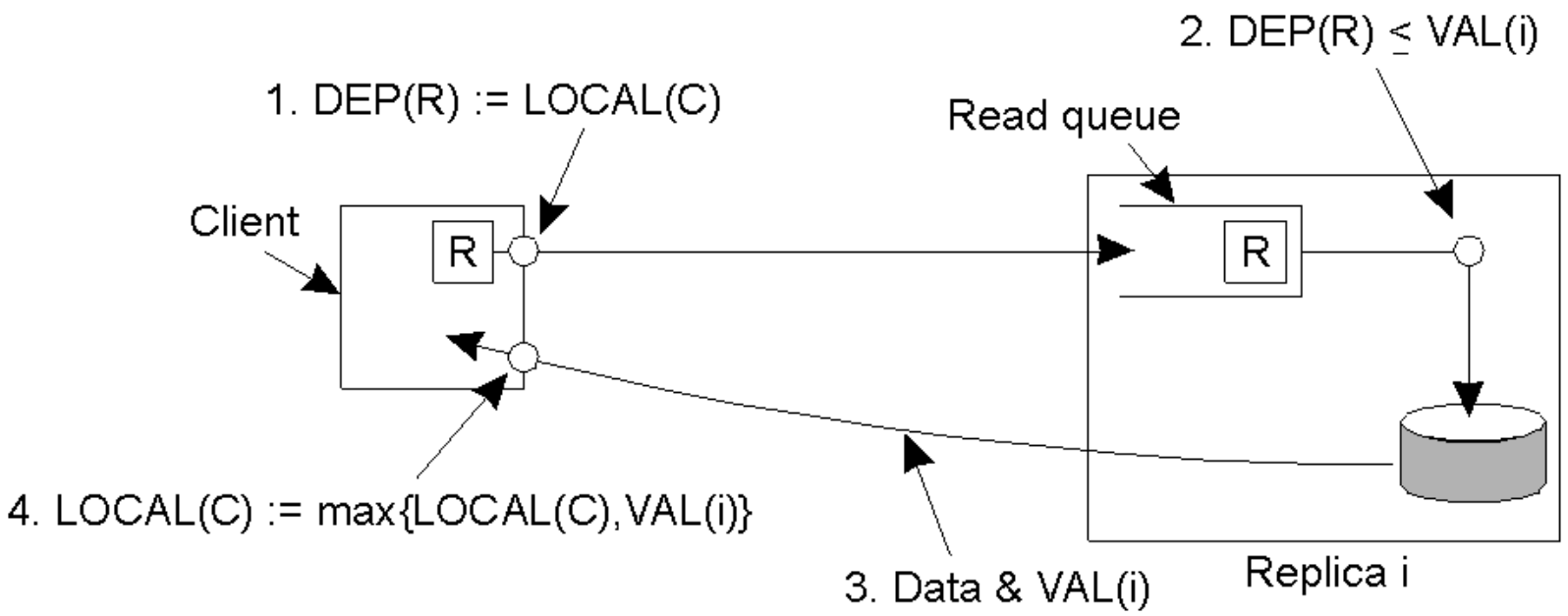
Regular 2PC implemented

# Causally-Consistent Lazy Replication

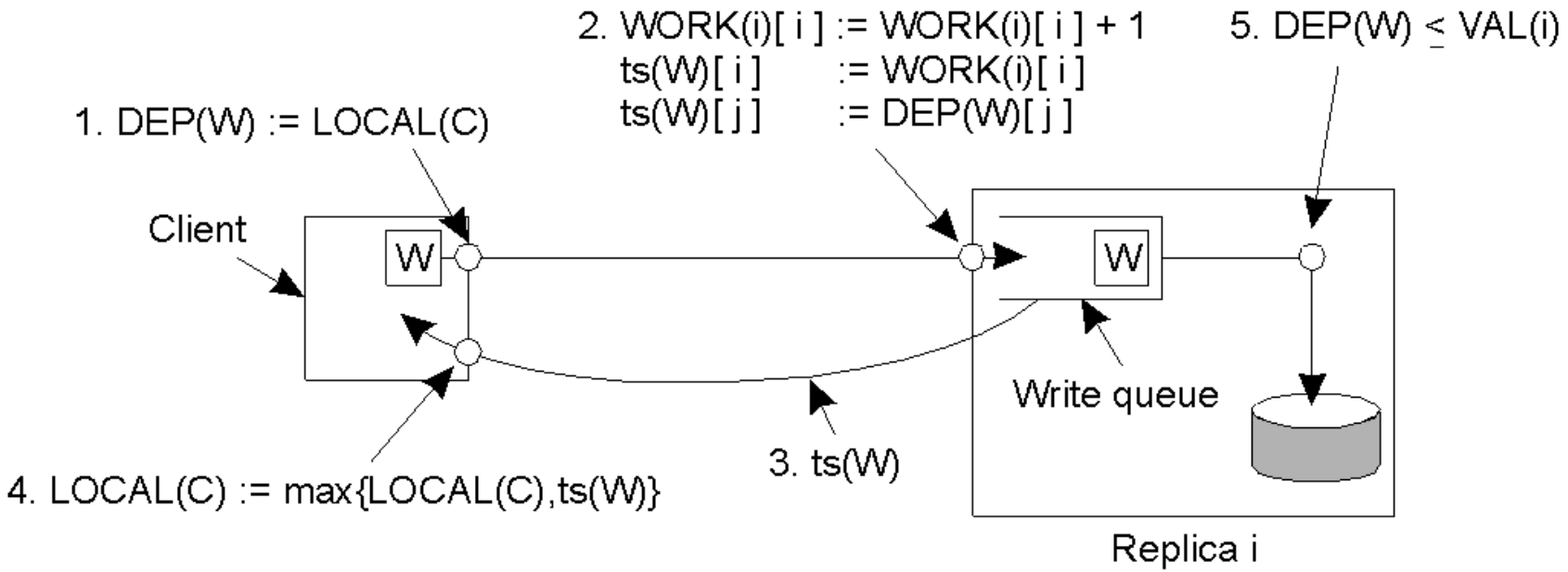



- The general organization of a distributed data store. Clients are assumed to also handle consistency-related communication.

# Processing Read Operations



# Processing Write Operations




- Performing a write operation at a local copy.